Implementarea unui Chatbot folosind Python și Google Search

**Ce este un chatbot?**

Un chatbot este un software software bazat pe inteligență artificială, care încearcă să înțeleagă nevoile utilizatorului, oferindu-i suport pentru a îndeplini o anumită sarcină, precum o tranzacție comercială, o rezervare la hotel, depunerea unui formular etc. [1]

**Evoluția chatbot-ului**

Istoria chatbot-urilor datează din 1966, când un program de calculator numit ELIZA a fost inventat de Weizenbaum, pentru a demonstra că comunicarea dintre om și mașină este superficială. ELIZA imită limbajul unui psihoterapeut și a fost dezvoltat urmând câteva din principiile fundamentale de proiectare care sunt folosite și astăzi. Printre aceste principii se enumeră:

* Botul își așteaptă rândul în conversație până când utilizatorul îi răspunde
* Botul nu are nevoie de o bază de date de cunoștințe din lumea reală pentru a comunica, afară de cunoștințele legate de problemele psihiatrice [2]

Chatbot-urile sunt, astăzi, în continuă evoluție, datorită dezvoltării tehnologiei de prelucrare a limbajului natural care au permis acestora să poarte conversații aproape umane.

Unul dintre cele mai interesante moduri prin care vedem că evoluează chatbots este prin integrare. În mod izolat, un chatbot este doar un set de răspunsuri programate. Cu ajutorul integrării, un chatbot poate accesa date istorice pentru a se asigura că toate întrebările sunt relevante pentru interogarea respectivului client, eliminând astfel întrebările inutile sau redundante și continuând să ofere o experiență mai plăcută. Mai mult, analiza sentimentelor poate ajuta robotii să detecteze când un client devine frustrat, ceea ce indică necesitatea unui transfer către un reprezentant al unui centru de contact. De asemenea, chatbot-urile pot fi astăzi integrați cu diferite sisteme sau baze de date interne, optimizând utilitatea acestora și făcându-le mai inteligente, prin automatizarea anumitor servicii oferite clienților, precum actualizarea detaliilor de plăți, crearea programărilor, etc. [3]

**Cum funcționează un chatbot?**

Există două tipuri de chatbot-uri: [4]

1. cei bazați pe reguli – Un bot răspunde la întrebări pe baza unor reguli pe care este instruit. Regulile definite pot fi foarte simple până la foarte complexe.
2. cei bazați pe învățare automată – utilizează unele abordări bazate pe Machine Learning și sunt cu siguranță mai eficienți decât bot-urile bazate pe reguli.

Modul de funcționare al unui chatbot este descris în diagrama următoare: [5]

**Cum se implementează un chatbot?**

Pentru a putea construi un chatbot, avem nevoie de cunoștințe de NLP (Natural Language Processing). NLP este domeniul de studiu care se concentrează pe interacțiunile dintre limbajul uman și computere. Acesta este o modalitate prin care calculatoarele pot analiza, înțelege și deriva sensul limbajului uman într-un mod inteligent și util. Limbajul de programare python oferă o librarie numită NLTK (Natural Language Toolkit), care vine în ajutorul programatorilor ce doresc să construiască chatbots inteligenți, bazați pe inteligență artificială. Librăria pune la dispoziție o suită de biblioteci de procesare a textului pentru clasificare, tokenizare, stemming, tagging, parsing și raționament semantic. [4]

**Ce trebuie să știm înainte de a implementa un chatbot?**

Înainte de a construi un chatbot, trebuie să ținem cont de anumite reguli, pentru ca acesta să fie o experiență placută pentru utilizatori.

1. Botul trebuie să cunoască informații despre utilizatorul cu care interacționează, pentru a putea formula răspunsurile potrivite. De exemplu, dacă oferim informații sportive, va fi de ajutor să se interogheze utilizatorul la începutul conversației cu privire la echipa favorită.
2. Programarea botului ca acesta să fie cât mai generic atunci când nu este sigur de răspuns. De asemenea, trebuie să folosească cuvinte cât mai simple.
3. Botul trebuie să conducă și să inițieze conversația. De exemplu, în momentul în care utilizatorul deschide fereastra de chat și spune salut botul va folosi această ocazie pentru a începe o conversație nouă și a oferi îndrumări.
4. Folosirea principiilor de comunicare. Botul nu trebuie sa arunce multe mesaje lungi și să folosească pauze între mesaje, pentru a permite utilizatorului să citească informațiile. De asemenea, mesajele trimise de către chatbot-uri ar trebui să încurajeze utilizatorii să-l folosească.
5. Atunci când botul trimite un răspuns, să fie formulat în așa fel încât conversația să continue.
6. Vizarea publicului potrivit. Atunci când introducem un bot nou într-o aplicație, trebuie să știm cu ce categorie de utilizatori va interacționa. De exemplu, tonul și limbajul botului ar putea fi diferit în funcție de categoriile de vârstă ale utilizatorilor. [6]

# Build a Simple ChatBot with Python and Google Search [7]

Below we are going to build a Python 3 ChatBot API and web interface. ChatBots are challenging to build because there are an infinite number of inputs. Because of that, a ChatBot that can consistently come up with good answers needs immense knowledge.

It is common for developers to apply machine learning algorithms, NLP, and corpora of predefined answers into their ChatBot system design. We are going to keep our code basic, so we will bypass creating a complex “brain” for our ChatBot.

Instead of building an AI brain, we will use one that is free and already built: Google Search.

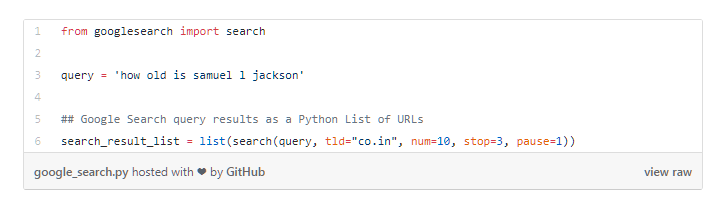
Our ChatBot will perform a Google Search of a user’s query, scrape the text from the first result, and reply to the user with the first sentence of that page’s text.

Let’s get started! By the way, all the code mentioned is in the Python ChatBot GitHub repository.

Querying Google In Python for ChatBot Replies

In order to program our simple ChatBot with omniscience (infinite knowledge), we will do Google searches within the Python API. Fortunately there is a Google search Python library that we can install with pip.

After you have installed the Google library locally, you can write Python code like this:



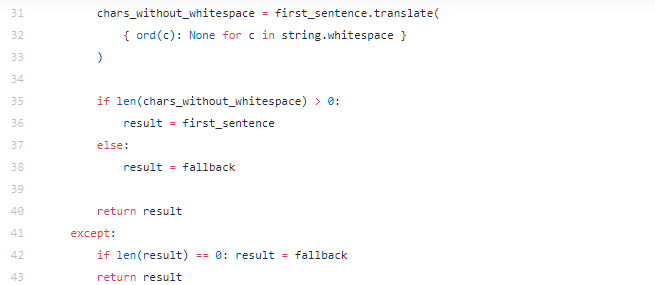
Once we have a list of URLs from the search results, we can do a GET request of that web page using the Python requests library. We can also parse the HTML by using html from lxml and also BeautifulSoup.



All of the Python dependencies of this project can be found in the requirements.txt file in the GitHub repository.

Here is a completed file that our HTTP server can import as a dependency. I made a method that does a Google search, gets the first <p> on the web page, and returns its contents as a string. If the search fails in any way, the ChatBot will reply with “Sorry, I cannot think of a reply for that.”

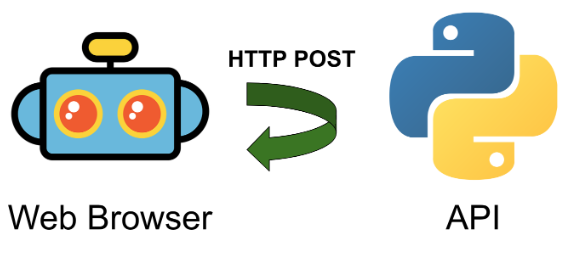




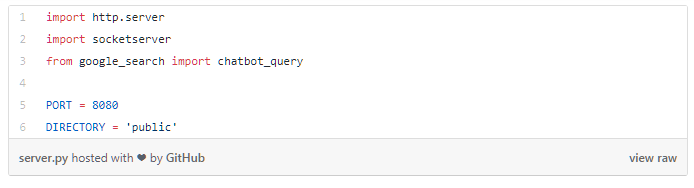
Now we can accept user input and do a Google search. We’ll make a HTTP GET request to the first result of the search. Then we parse through the HTML that was returned and isolate the first sentence in the first <p> on that page. This is our ChatBot’s reply algorithm, no machine learning required.

Python API for a Simple ChatBot

Next we need to build a server app that will be our API for ChatBot queries. It will serve responses to HTTP requests. To start, those requests will come from a simple HTML page which we’ll make later.

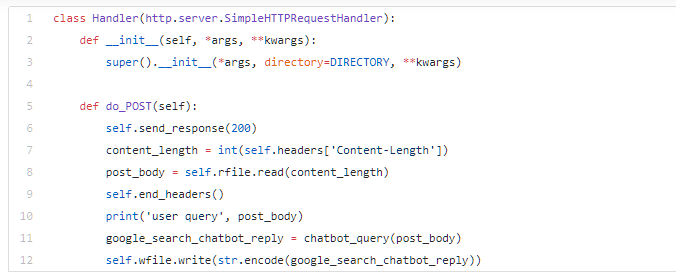


To start, we will import the Python 3 HTTP server and socket server libraries along with the Google search file we made earlier.



Our API will be served on port 8080 and we will serve web page assets from a folder called public in our project’s parent directory. Next we will make our own handler for GET and POST requests.

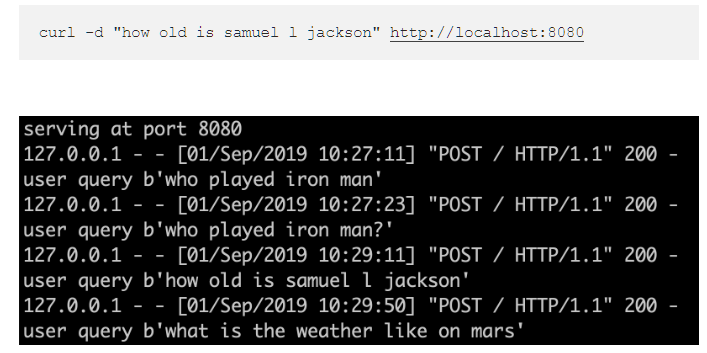
HTTP GET requests will attempt to return a corresponding file from the public folder. These will be the HTML, CSS, and JavaScript files for our web browser interface. POST requests will be used for ChatBot queries.



Lastly, we will start up the server and use our handler. Here is the entire file, including the above code snippets.



We can use CURL to test out the ChatBot API with POST requests.



Next we will make an HTML page that can query this API. By the end, we’ll have an end-to-end ChatBot that provides sophisticated answers.

Building the ChatBot Web Page

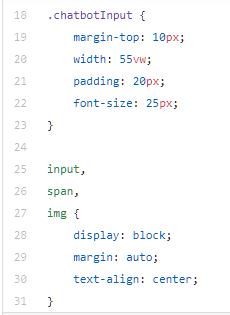
Our web page will be very simple. It will contain a picture of a bot, a text input field, and a submit button. Whenever the user submits an input, the chatbot API will be reached through a POST request. The text answer that is returned from the API will be filled-in on the web page.

Here is the HTML page. Save this as index.html in the public folder we mentioned earlier. The image file of the bot is also in the full Python ChatBot GitHub repository.



Next we will do some quick styles for this web page. Save this CSS file in the public folder too. It is already referenced in the HTML file’s <head> section.





The web page is not quite ready for users. It needs JavaScript.

We’ll write some JS that detects a user pressing the Return key and also clicking the submit button. When either of those events happen, we’ll get the text inside the user input field and include it as a POST body for our Python server.

We’ll make a HTTP POST request to the Python API server using the fetch method. The Fetch API is now included by default in modern web browsers.

Here is our simple JavaScript that interacts with our 3 HTML elements. Save this as app.js in the public folder.



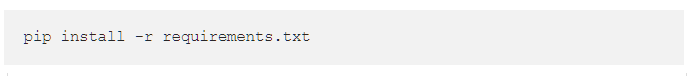


We’re almost ready to ship our Python ChatBot to production.

Running our Simple Python ChatBot Made From Scratch

Now that we have written all the code, we have one more step before we can run the full-stack Python ChatBot. If you have not already, make a requirements.txt file in the parent directory of the project, alongside the 2 Python files. This file is a Python paradigm for easily installing a project’s dependencies.



Go to the parent directory of the ChatBot project using your command line. run the Python library install command.

Your machine now has all the required libraries to run the ChatBot! Let’s run the full-stack app.



Next open your web browser and go to http://localhost:8080/. If you see the ChatBot image, it is working. You can see, our ChatBot replies aren’t perfect, but pretty good for a few minutes of work.

# How to create an intelligent chatbot in Python[8]

Natural language processing (NLP) is one of the most promising fields of artificial intelligence that uses natural languages to enable human interactions with machines.

There are two main approaches to NLP:

* rule-based methods,
* statistical methods, i.e., methods related to machine learning.

There are several exciting Python libraries for NLP, such as Natural Language Toolkit (NLTK), spaCy, TextBlob, etc.

A chatbot is a computer software able to interact with humans using a natural language. They usually rely on machine learning, especially on NLP. Apple’s Siri, Amazon’s Alexa, Google Assistant, and Microsoft’s Cortana are some well-known examples of software able to process natural languages.

This article shows how to create a simple chatbot in Python using the library ChatterBot. Our bot will be used for small talk, as well as to answer some math questions. Here, we’ll scratch the surface of what’s possible in building custom chatbots and NLP in general.

**Preparing Dependencies**

You’re only going to install the library ChatterBot for now. I recommend creating and using a new Python virtual environment for this purpose. Execute the following commands in your Python (or Anaconda) terminal:

* pip install chatterbot
* pip install chatterbot\_corpus

You can also try upgrading them:

* pip install --upgrade chatterbot\_corpus
* pip install --upgrade chatterbot

That’s it. We’re ready to go.

**Importing Classes**

You’ll need to import two classes for this purpose: ChatBot from chatterbot and ListTrainer from chatterbot.trainers:

* from chatterbot import ChatBot
* from chatterbot.trainers import ListTrainer

We’re now prepared to create and train our math bot.

**Creating and Training a Bot**

Our bot will be an instance of the class ChatBot:

* my\_bot = ChatBot(name='PyBot', read\_only=True,
  + logic\_adapters=['chatterbot.logic.MathematicalEvaluation',
    - 'chatterbot.logic.BestMatch'])

The only required argument corresponds to the parameter name. It represents the name of the bot. You can provide read\_only=True if you want to disable the bot’s ability to learn after the training (i.e. from actual conversations). logic\_adapters is the list of adapters used to train the bot. There are several of them provided by chatterbot, like the two from our example. chatterbot.logic.MathematicalEvaluation enables the bot to solve math problems, while chatterbot.logic.BestMatch chooses the best match from the already provided responses.

So, we have to provide responses. We do that by specifying the lists of strings later used to train the bot and find the best match for each question. This is what I want our bot to learn for now:

small\_talk = ['hi there!',  
 'hi!',  
 'how do you do?',  
 'how are you?',  
 'i\'m cool.',  
 'fine, you?',  
 'always cool.',  
 'i\'m ok',  
 'glad to hear that.',  
 'i\'m fine',  
 'glad to hear that.',  
 'i feel awesome',  
 'excellent, glad to hear that.',  
 'not so good',  
 'sorry to hear that.',  
 'what\'s your name?',  
 'i\'m pybot. ask me a math question, please.']math\_talk\_1 = ['pythagorean theorem',  
 'a squared plus b squared equals c squared.']math\_talk\_2 = ['law of cosines',  
 'c\*\*2 = a\*\*2 + b\*\*2 - 2 \* a \* b \* cos(gamma)']

We can create and train the bot by creating an instance of ListTrainer and supplying it with the lists of strings:

list\_trainer = ListTrainer(my\_bot)for item in (small\_talk, math\_talk\_1, math\_talk\_2):  
 list\_trainer.train(item)

The bot should now be trained and ready to communicate.

**Communicating with a Bot**

You can communicate with your bot using its method .get\_response(). Here’s an example of how that might look like:

>>> print(my\_bot.get\_response("hi"))  
how do you do?>>> print(my\_bot.get\_response("i feel awesome today"))  
excellent, glad to hear that. >>> print(my\_bot.get\_response("what's your name?"))  
i'm pybot. ask me a math question, please.>>> print(my\_bot.get\_response("show me the pythagorean theorem"))  
a squared plus b squared equals c squared.>>> print(my\_bot.get\_response("do you know the law of cosines?"))  
c\*\*2 = a\*\*2 + b\*\*2 - 2 \* a \* b \* cos(gamma)

Don’t expect the bot to answer each question well! Its knowledge is limited to the stuff similar to what it has learned. Many times, you’ll find it answering nonsense, especially if you don’t provide comprehensive training.

**Training a Bot with a Corpus of Data**

You can use your own or an existing corpus of data to train a bot. For example, you can use some corpus provided by chatterbot:

from chatterbot.trainers import ChatterBotCorpusTrainercorpus\_trainer = ChatterBotCorpusTrainer(my\_bot)  
corpus\_trainer.train('chatterbot.corpus.english')

chatterbot offers this functionality in several languages. You can also specify a subset of a corpus you’d like to use.

**Conclusion**

Now you know how to create and use a simple chat bot.

This is just a small illustration of what you can do with natural language processing and chatbots. There are many more possibilities out there. If you’re interested in exploring them, you can start by getting familiar with NLTK and ChatterBot.

# Python for NLP: Creating a Rule-Based Chatbot [9]

This is the 12th article in my series of articles on Python for NLP. In the previous article, I briefly explained the different functionalities of the Python's Gensim library. Until now, in this series, we have covered almost all of the most commonly used NLP libraries such as NLTK, SpaCy, Gensim, StanfordCoreNLP, Pattern, TextBlob, etc.

In this article, we are not going to explore any NLP library. Rather, we will develop a very simple rule-based chatbot capable of answering user queries regarding the sport of Tennis. But before we begin actual coding, let's first briefly discuss what chatbots are and how they are used.

**What is a Chatbot?**

A chatbot is a conversational agent capable of answering user queries in the form of text, speech, or via a graphical user interface. In simple words, a chatbot is a software application that can chat with a user on any topic. Chatbots can be broadly categorized into two types: Task-Oriented Chatbots and General Purpose Chatbots.

The task-oriented chatbots are designed to perform specific tasks. For instance, a task-oriented chatbot can answer queries related to train reservation, pizza delivery; it can also work as a personal medical therapist or personal assistant.

On the other hand, general purpose chatbots can have open-ended discussions with the users.

There is also a third type of chatbots called hybrid chatbots that can engage in both task-oriented and open-ended discussion with the users.

**Approaches for Chatbot Development**

Chatbot development approaches fall in two categories: rule-based chatbots and learning-based chatbots.

**Learning-Based Chatbots**

Learning-based chatbots are the type of chatbots that use machine learning techniques and a dataset to learn to generate a response to user queries. Learning-based chatbots can be further divided into two categories: retrieval-based chatbots and generative chatbots.

The retrieval based chatbots learn to select a certain response to user queries. On the other hand, generative chatbots learn to generate a response on the fly.

One of the main advantages of learning-based chatbots is their flexibility to answer a variety of user queries. Though the response might not always be correct, learning-based chatbots are capable to answer to any type of user query. One of the major drawbacks of these chatbots is that they may need a huge amount of time and data to train.

**Rule-Based Chatbots**

Rule-based chatbots are pretty straight forward as compared to learning-based chatbots. There are a specific set of rules. If the user query matches any rule, the answer to the query is generated, otherwise the user is notified that the answer to user query doesn't exist.

One of the advantages of rule-based chatbots is that they always give accurate results. However, on the downside, they do not scale well. To add more responses, you have to define new rules.

In the following section, I will explain how to create a rule-based chatbot that will reply to simple user queries regarding the sport of tennis.

**Rule-Based Chatbot Development with Python**

The chatbot we are going to develop will be very simple. First we need a corpus that contains lots of information about the sport of tennis. We will develop such a corpus by scraping the Wikipedia article on tennis. Next, we will perform some preprocessing on the corpus and then will divide the corpus into sentences.

When a user enters a query, the query will be converted into vectorized form. All the sentences in the corpus will also be converted into their corresponding vectorized forms. Next, the sentence with the highest cosine similarity with the user input vector will be selected as a response to the user input.

Follow these steps to develop the chatbot:

**Importing Required Libraries**

import nltk

import numpy as np

import random

import string

import bs4 as bs

import urllib.request

import re

We will be using the Beautifulsoup4 library to parse the data from Wikipedia. Furthermore, Python's regex library, re, will be used for some preprocessing tasks on the text.

**Creating the Corpus**

As we said earlier, we will use the Wikipedia article on Tennis to create our corpus. The following script retrieves the Wikipedia article and extracts all the paragraphs from the article text. Finally the text is converted into the lower case for easier processing.

raw\_html = urllib.request.urlopen('https://en.wikipedia.org/wiki/Tennis') raw\_html = raw\_html.read() article\_html = bs.BeautifulSoup(raw\_html, 'lxml') article\_paragraphs = article\_html.find\_all('p') article\_text = '' for para in article\_paragraphs: article\_text += para.text article\_text = article\_text.lower()

**Text Preprocessing and Helper Function**

Next, we need to preprocess our text to remove all the special characters and empty spaces from our text. The following regular expression does that:

article\_text = re.sub(r'\[[0-9]\*\]', ' ', article\_text) article\_text = re.sub(r'\s+', ' ', article\_text)

We need to divide our text into sentences and words since the cosine similarity of the user input will actually be compared with each sentence. Execute the following script:

article\_sentences = nltk.sent\_tokenize(article\_text) article\_words = nltk.word\_tokenize(article\_text)

Finally, we need to create helper functions that will remove the punctuation from the user input text and will also lemmatize the text. Lemmatization refers to reducing a word to its root form. For instance, lemmatization the word "ate" returns eat, the word "throwing" will become throw and the word "worse" will be reduced to "bad".

Execute the following code:

wnlemmatizer = nltk.stem.WordNetLemmatizer() def perform\_lemmatization(tokens): return [wnlemmatizer.lemmatize(token) for token in tokens] punctuation\_removal = dict((ord(punctuation), None) for punctuation in string.punctuation) def get\_processed\_text(document): return perform\_lemmatization(nltk.word\_tokenize(document.lower().translate(punctuation\_removal)))

In the script above we first instantiate the WordNetLemmatizer from the NTLK library. Next, we define a function perform\_lemmatization, which takes a list of words as input and lemmatize the corresponding lemmatized list of words. The punctuation\_removal list removes the punctuation from the passed text. Finally, the get\_processed\_text method takes a sentence as input, tokenizes it, lemmatizes it, and then removes the punctuation from the sentence.

**Responding to Greetings**

Since we are developing a rule-based chatbot, we need to handle different types of user inputs in a different manner. For instance, for greetings we will define a dedicated function. To handle greetings, we will create two lists: greeting\_inputs and greeting\_outputs. When a user enters a greeting, we will try to search it in the greetings\_inputs list, if the greeting is found, we will randomly choose a response from the greeting\_outputs list.

Look at the following script:

greeting\_inputs = ("hey", "good morning", "good evening", "morning", "evening", "hi", "whatsup") greeting\_responses = ["hey", "hey hows you?", "\*nods\*", "hello, how you doing", "hello", "Welcome, I am good and you"]

def generate\_greeting\_response(greeting):

for token in greeting.split():

if token.lower() in greeting\_inputs:

return random.choice(greeting\_responses)

Here the generate\_greeting\_response() method is basically responsible for validating the greeting message and generating the corresponding response.

**Responding to User Queries**

As we said earlier, the response will be generated based upon the cosine similarity of the vectorized form of the input sentence and the sentences in the corpora. The following script imports the TfidfVectorizer and the cosine\_similarity functions:

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

Now we have everything set up that we need to generate a response to the user queries related to tennis. We will create a method that takes in user input, finds the cosine similarity of the user input and compares it with the sentences in the corpus.

Look at the following script:

def generate\_response(user\_input): tennisrobo\_response = '' article\_sentences.append(user\_input) word\_vectorizer = TfidfVectorizer(tokenizer=get\_processed\_text, stop\_words='english') all\_word\_vectors = word\_vectorizer.fit\_transform(article\_sentences) similar\_vector\_values = cosine\_similarity(all\_word\_vectors[-1], all\_word\_vectors) similar\_sentence\_number = similar\_vector\_values.argsort()[0][-2] matched\_vector = similar\_vector\_values.flatten() matched\_vector.sort() vector\_matched = matched\_vector[-2] if vector\_matched == 0: tennisrobo\_response = tennisrobo\_response + "I am sorry, I could not understand you" return tennisrobo\_response else: tennisrobo\_response = tennisrobo\_response + article\_sentences[similar\_sentence\_number] return tennisrobo\_response

You can see that the generate\_response() method accepts one parameter which is user input. Next, we define an empty string tennisrobo\_response. We then append the user input to the list of already existing sentences. After that in the following lines:

word\_vectorizer = TfidfVectorizer(tokenizer=get\_processed\_text, stop\_words='english') all\_word\_vectors = word\_vectorizer.fit\_transform(article\_sentences)

We initialize the tfidfvectorizer and then convert all the sentences in the corpus along with the input sentence into their corresponding vectorized form.

In the following line:

similar\_vector\_values = cosine\_similarity(all\_word\_vectors[-1], all\_word\_vectors)

We use the cosine\_similarity function to find the cosine similarity between the last item in the all\_word\_vectors list (which is actually the word vector for the user input since it was appended at the end) and the word vectors for all the sentences in the corpus.

Next, in the following line:

similar\_sentence\_number = similar\_vector\_values.argsort()[0][-2]

We sort the list containing the cosine similarities of the vectors, the second last item in the list will actually have the highest cosine (after sorting) with the user input. The last item is the user input itself, therefore we did not select that.

Finally, we flatten the retrieved cosine similarity and check if the similarity is equal to zero or not. If the cosine similarity of the matched vector is 0, that means our query did not have an answer. In that case, we will simply print that we do not understand the user query.

Otherwise, if the cosine similarity is not equal to zero, that means we found a sentence similar to the input in our corpus. In that case, we will just pass the index of the matched sentence to our "article\_sentences" list that contains the collection of all sentences.

**Chatting with the Chatbot**

As a final step, we need to create a function that allows us to chat with the chatbot that we just designed. To do so, we will write another helper function that will keep executing until the user types "Bye".

Look at the following script, the code has been explained after that:

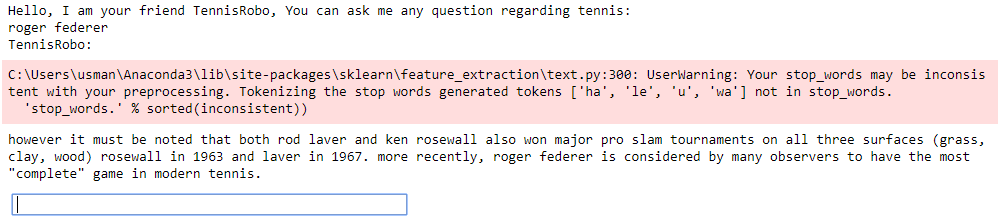
continue\_dialogue = True print("Hello, I am your friend TennisRobo. You can ask me any question regarding tennis:") while(continue\_dialogue == True): human\_text = input() human\_text = human\_text.lower() if human\_text != 'bye': if human\_text == 'thanks' or human\_text == 'thank you very much' or human\_text == 'thank you': continue\_dialogue = False print("TennisRobo: Most welcome") else: if generate\_greeting\_response(human\_text) != None: print("TennisRobo: " + generate\_greeting\_response(human\_text)) else: print("TennisRobo: ", end="") print(generate\_response(human\_text)) article\_sentences.remove(human\_text) else: continue\_dialogue = False print("TennisRobo: Good bye and take care of yourself...")

In the script above, we first set the flag continue\_dialogue to true. After that, we print a welcome message to the user asking for any input. Next, we initialize a while loop that keeps executing until the continue\_dialogue flag is true. Inside the loop, the user input is received, which is then converted to lower case. The user input is stored in the human\_text variable. If the user enters the word "bye", the continue\_dialogue is set to false and goodbye message is printed to the user.

On the other hand, if the input text is not equal to "bye", it is checked if the input contains words like "thanks", "thank you", etc. or not. If such words are found, a reply "Most welcome" is generated. Otherwise, if the user input is not equal to None, the generate\_response method is called which fetches the user response based on the cosine similarity as explained in the last section.

Once the response is generated, the user input is removed from the collection of sentences since we do not want the user input to be part of the corpus. The process continues until the user types "bye". You can see why this type of chatbot is called a rule-based chatbot. There are plenty of rules to follow and if we want to add more functionalities to the chatbot, we will have to add more rules.

The output of the chatbot script looks like this:



You can see in the above image that I entered the input "roger federer" and the response generated is:

however it must be noted that both rod laver and ken rosewall also won major pro slam tournaments on all three surfaces (grass, clay, wood) rosewall in 1963 and laver in 1967. more recently, roger federer is considered by many observers to have the most "complete" game in modern tennis."

The response might not be precise, however, it still makes sense.

It is important to mention that the idea of this article is not to develop a perfect chatbot but to explain the working principle of rule-based chatbots.

**Conclusion**

Chatbots are conversational agents that engage in different types of conversations with humans. Chatbots are finding their place in different strata of life ranging from personal assistant to ticket reservation systems and physiological therapists. Having a chatbot in place of humans can actually be very cost effective. However, developing a chatbot with the same efficiency as humans can be very complicated.

# Referințe

|  |  |
| --- | --- |
| [1] | H. Pelk, "Natural Language Processing and Machine Learning," 10 Februarie 2017. [Online]. Available: https://chatbotsmagazine.com/natural-language-processing-and-machine-learning-the-core-of-the-modern-smart-chatbot-8755c6343fa5. |
| [2] | S. Wolter, "Eliza, a Chatbot as a Psychotherapist," 25 Iunie 2019. [Online]. Available: https://www.linkedin.com/pulse/eliza-chatbot-psychotherapist-sascha-wolter/. |
| [3] | T. Jamous, "Chatbot: The revolution and evolution of the AI-powered technology," [Online]. Available: https://www.comparethecloud.net/articles/chatbot-revolution-evolution-ai/. |
| [4] | P. Pandey, "Building a Simple Chatbot from Scratch in Python (using NLTK)," 17 Septembrie 2018. [Online]. Available: https://medium.com/analytics-vidhya/building-a-simple-chatbot-in-python-using-nltk-7c8c8215ac6e. |
| [5] | L. Kim, "How Do Chatbots Work? A Practical Business Guide to Building Chatbots for Sales, Marketing and Customer Service," 25 Septembrie 2019. [Online]. Available: https://medium.com/marketing-and-entrepreneurship/how-do-chatbots-work-459353cb8895. |
| [6] | J. Monteon, "6 User Experience Tips for Designing Your Best Chatbot," 18 Noiembrie 2016. [Online]. Available: https://chatbotsmagazine.com/6-tips-for-designing-your-best-chatbot-591aba9c9eff. |
| [7] | L. Zach, "Build a Simple ChatBot with Python and Google Search," 2 Septembrie 2019. [Online]. Available: https://towardsdatascience.com/build-a-simple-chatbot-with-python-and-google-search-c000aa3f73f0. |
| [8] | R. Fabisiak, "How to create an intelligent chatbot in Python," 27 Iunie 2019. [Online]. Available: https://chatbotslife.com/how-to-create-an-intelligent-chatbot-in-python-c655eb39d6b1. |
| [9] | U. Malik, "Python for NLP: Creating a Rule-Based Chatbot," 28 Iunie 2019. [Online]. Available: https://stackabuse.com/python-for-nlp-creating-a-rule-based-chatbot/. |